Performance

Play+ Performance Helper : playperf Introduction ■ In the Play+ ecosystem, performance is not a feature to be added later—it's a foundational commitment. We believe great design must feel fast. This helper is based on the concept of Performance by Default, where speed, responsiveness, and stability are architected and enforced from day one. A performant application builds user trust and delight. This directly supports our core design pillars by creating an Engaging experience that feels fluid and responsive, and an Adaptive interface that performs well across all devices and network conditions. By ensuring a fast, stable experience, we also make our products more Intuitive and less frustrating to use. Package Info ■ The Play+ performance helper and its associated CI configurations are included by default in the Golden Path starter kit. Description ■ Package / Path Golden Path (Recommended) Pre-installed /system/play.perf.ts Uplift Path npm install @playplus/perf Folder Reference ■ The performance helper and its configuration follow our standardized folder structure for core system logic. File / Directory Purpose & Guidelines system/play.perf.ts The core performance helper. It provides utilities for deferring tasks and monitoring real-world performance. config/play.performance.config.json User-overridable configuration for performance budgets (Lighthouse) and monitoring settings. reports/performance/ The git-ignored directory where Lighthouse CI reports are saved for local inspection. Helper - Pillars Alignment ■ The playperf helper is a direct implementation of our core design pillars, focused on the user's perception of speed. Pillar How This Helper Aligns Engaging Primary Pillar: Ensures snappy interactions and fluid animations, which are key to a delightful and engaging user experience. Adaptive Enforces performance budgets that ensure the application is fast and responsive on a wide range of devices and networks. Intuitive A fast and stable application feels more predictable and intuitive, as users aren't left waiting or dealing with layout shifts. Helper Overview ■ The playperf toolchain is a combination of automated guardrails and developer-facing utilities designed to abstract the plumbing of performance management. Instead of manually configuring performance budgets or writing complex code to defer non-critical tasks, developers can rely on the Play+ system to handle it. It automates performance quality control in two key ways: Preventative Guardrails : In the CI/CD pipeline, playperf automatically runs Lighthouse audits against every pull request. If key metrics (like LCP or CLS) exceed the configured budgets, the build fails, preventing performance regressions from ever reaching production. Developer Empowerment Utilities : It provides a simple runtime helper with methods like defer() and monitor() that make it trivial to implement common performance patterns without writing boilerplate code. The goal is to make high performance the default path, not an extra chore. Config Options ■ Global performance budgets and settings are managed in config/play.performance.config.json . These values are consumed by the CI pipeline and runtime helpers. Config Key Table ■ Config Key Default Value Description Recommended Value lighthouse.lcp 2500 Lighthouse budget for Largest Contentful Paint (ms). 2500 lighthouse.cls 0.1 Lighthouse budget for Cumulative Layout Shift. 0.1 lighthouse.inp 200 Lighthouse budget for Interaction to Next Paint (ms). 200

lighthouse.maxBundleSize 250 Lighthouse budget for initial JS bundle size (KB, gzipped). 250 enforce.blockBuildOnFail true If true, the CI build will fail if performance budgets are exceeded. true enforce.enableBundleDiff true If true, PRs will be flagged with any unexpected changes to bundle size. true monitor.webVitals true If true, enables the real-user monitoring of Core Web Vitals via playperf.monitor . true monitor.sentry true Enables integration with Sentry Performance for transaction tracking. true assist.highlightHeavyComponents true In dev mode, flags components that are computationally expensive or re-rendering unnecessarily. true Helper Methods ■ Core Methods ■ Method Name Description Signature defer Schedules non-critical tasks to run during browser idle time, preventing them from blocking critical rendering. defer(callback: () => void, options?: { priority: 'high' | 'low', timeout: number }): void monitor Initializes comprehensive performance monitoring including Core Web Vitals, long tasks, and bundle size tracking. monitor(options: { onReport: (metric) => void, enableWebVitals?: boolean, enableLongTasks?: boolean, enableBundleMonitoring?: boolean }): void measure Measures the performance of synchronous functions. measure<T>(name: string, fn: () => T): T measureAsync Measures the performance of asynchronous functions. measureAsync<T>(name: string, fn: () => Promise<T>): Promise<T> Angular Integration ■ PlayPerfService ■ Angular service wrapper that integrates with Play+ logging and provides component-specific performance utilities. import { PlayPerfService } from '../services/playperf.service' ; @ Component ({ ... }) export class MyComponent { constructor (private playperf : PlayPerfService) { } ngOnInit ( ) { // Initialize monitoring this . playperf . monitor ( ) ; } processData ( ) { // Measure expensive operations return this . playperf . measure ('dataProcessing' , ( ) => { return this . heavyComputation ( ) ; }) ; } } Performance Directive ■ Automatically monitors component render times and highlights heavy components. <!-- Monitor component performance --> < div playPerf = " user-list " perfThreshold = " 16 " > < app-user-list [users] = " users " > </ app-user-list > </ div > Performance Pipe ■ Measures and reports template expression performance. <!-- Monitor expensive template expressions --> < div > </ div > React: Deferring a Non-Critical Script ■ This example shows how to defer the initialization of an analytics script so it doesn't interfere with the initial page load. import React , { useEffect } from 'react' ; import { playperf } from '../../system/play.perf' ; import { initializeAnalytics } from '../lib/analytics' ; function App ( ) { useEffect (( ) => { // Defer the analytics script to run after the main thread is free. playperf . defer (( ) => { initializeAnalytics ({ apiKey : '...' }) ; }) ; } , [ ]) ; return (// ... your application components) ; } Angular: Monitoring Real-World Performance ■ This example shows how to initialize Web Vitals monitoring in your main application component and log the results. // app.component.ts import { Component , OnInit } from "@angular/core" ; import { playperf } from "@playplus/core" ; // Assuming helper is available import { playlog } from "@playplus/core" ; @ Component ({ selector : "app-root" , templateUrl : "./app.component.html" , }) export class AppComponent implements OnInit { ngOnInit ( ) { // Start monitoring real-user performance metrics. playperf . monitor ({ onReport : (metric) => { // Send the collected metric to our logging service. playlog . info (` Web Vitals Metric: ${ metric . name } ` , { value : metric . value , id : metric . id , }) ; } , }) ; } } import { playperf } from "../../../system/play.perf" ; // Defer non-critical initialization playperf . defer (( ) => {

initializeAnalytics ( ) ; loadNonCriticalScripts ( ) ; }) ; // High priority defer (shorter timeout) playperf . defer (( ) => { updateUserPreferences ( ) ; } , { priority : "high" , timeout : 100 }) ; // Measure sync function const result = playperf . measure ("dataProcessing" , ( ) => { return processLargeDataset (data) ; }) ; // Measure async function const result = await playperf . measureAsync ("apiCall" , async ( ) => { return await fetchUserData ( ) ; }) ; Additional Info ■ Why We Created This Helper ■ Web performance is critical, but its tooling is complex. Without a dedicated system, each team would need to: Manually set up, configure, and maintain Lighthouse CI for every project. Write custom logic to track bundle sizes. Implement their own utilities for deferring scripts or monitoring Web Vitals. This is inefficient and leads to inconsistent standards. The playperf helper automates the enforcement and simplifies the implementation of performance best practices. It provides a pre-configured safety net and easy-to-use tools so developers can build fast experiences by default. Performance Budgets ■ All Play+ applications must meet these baseline performance standards: Metric Target Budget LCP (Largest Contentful Paint) < 2.5s 2500ms INP (Interaction to Next Paint) < 200ms 200ms CLS (Cumulative Layout Shift) < 0.1 0.1 JS Bundle Size (gzipped) < 250KB 250KB Main Thread Long Tasks (>50ms) 0 0 Best Practices ■ Change Detection Strategy ■ Use OnPush change detection for components that don't need frequent updates: @ Component ({ changeDetection : ChangeDetectionStrategy . OnPush , // ... }) export class OptimizedComponent { // Component logic } TrackBy Functions ■ Always provide trackBy functions for ngFor loops: @ Component ({ ... }) export class ListComponent { trackByFn (index : number , item : any) : any { return item . id || index ; } } < div = " let item of items; trackBy: trackByFn " > </ div > Lazy Loading ■ Use lazy loading for routes and components: const routes : Routes = [{ path : "dashboard" , loadComponent : ( ) => import ("./dashboard/dashboard.component") . then ((m) => m . DashboardComponent) , } ,] ; Data Management ■ Caching Strategy ■ Use Play+ caching for expensive operations: import { PlayCacheService } from "../services/playcache.service" ; @ Injectable ( ) export class DataService { constructor (private playcache : PlayCacheService) { } async getUsers ( ) : Promise < User [ ] > { return this . playcache . get ("users" , async ( ) => { return this . apiService . fetchUsers ( ) ; } , { ttl : 300 }) ; // 5 minutes cache } } Virtual Scrolling ■ Use virtual scrolling for large lists: import { ScrollingModule } from "@angular/cdk/scrolling" ; @ Component ({ imports : [ScrollingModule] , // ... }) export class LargeListComponent { items = Array . from ({ length : 10000 } , (_ , i) => ` Item ${ i } `) ; } < cdk-virtual-scroll-viewport itemSize = " 50 " class = " list-container " > < div *cdkVirtualFor = " let item of items " > </ div > </ cdk-virtual-scroll-viewport > Image Optimization ■ Lazy Loading ■ Always lazy load images: < img [src] = " imageUrl " loading = " lazy " [alt] = " imageAlt " /> Modern Formats ■ Use WebP with fallbacks: < picture > < source [srcset] = " imageUrl + '.webp' " type = " image/webp " /> < img [src] = " imageUrl + '.jpg' " [alt] = " imageAlt " /> </ picture > Explicit Dimensions ■ Always specify width and height to prevent layout shifts: < img [src] = " imageUrl " width = " 300 " height = " 200 " [alt] = " imageAlt " /> Forbidden Patterns ■ Blocking Operations ■ // DON'T: Block main thread with heavy operations ngOnInit ( ) { this . heavyComputation ( ) ; // Blocks rendering } // DO: Defer non-critical operations ngOnInit ( ) { this . playperf . defer (( ) => { this . heavyComputation ( ) ; }) ; } Synchronous API Calls ■ //

DON'T: Make synchronous API calls getData ( ) { return this . http . get ('/api/data') . toPromise ( ) ; // Blocks UI } // DO: Use async/await with proper error handling async getData ( ) { return await this . playperf . measureAsync ('apiCall' , async ( ) => { return this . http . get ('/api/data') . toPromise ( ) ; }) ; } Expensive Template Expressions ■ <!-- DON'T: Expensive operations in templates --> < div > </ div > <!-- DO: Use pipes or computed properties --> < div > </ div > Manual setTimeout/setInterval ■ // DON'T: Use setTimeout directly setTimeout (( ) => { this . updateData ( ) ; } , 1000) ; // DO: Use playperf.defer() this . playperf . defer (( ) => { this . updateData ( ) ; } , { timeout : 1000 }) ; Required Patterns ■ Performance Monitoring ■ // Always initialize performance monitoring in app component ngOnInit ( ) { this . playperf . monitor ({ onReport : (metric) => { // Handle metrics appropriately } }) ; } // Use performance directive for heavy components @ Component ({ ... }) export class HeavyComponent { constructor (private playperf : PlayPerfService) { } ngOnInit ( ) { this . playperf . highlightHeavyComponent ('HeavyComponent' , 0) ; } } Async Data Loading ■ // Always use async patterns for data loading async loadData ( ) { return await this . playperf . measureAsync ('dataLoading' , async ( ) => { return this . apiService . getData ( ) ; }) ; } Testing Performance ■ Unit Testing ■ describe ("PerformanceService" , ( ) => { it ("should measure function performance" , ( ) => { const result = playperf . measure ("test" , ( ) => { return "test result" ; }) ; expect (result) . toBe ("test result") ; }) ; }) ; Integration Testing ■ describe ("Component Performance" , ( ) => { it ("should render within performance budget" , ( ) => { const start = performance . now ( ) ; fixture . detectChanges ( ) ; const renderTime = performance . now ( ) - start ; expect (renderTime) . toBeLessThan (16) ; // 60fps budget }) ; }) ; Monitoring and Analytics ■ Core Web Vitals ■ LCP : Largest Contentful Paint - measures loading performance FID : First Input Delay - measures interactivity CLS : Cumulative Layout Shift - measures visual stability Custom Metrics ■ Component render times API response times Bundle size changes Long task detection Reporting ■ Performance metrics are automatically logged and can be sent to: Play+ logging system Analytics platforms Monitoring dashboards CI/CD pipelines ### Lighthouse CI Configuration Example This is a sample of the configuration used by CI to assert performance budgets. It is managed by the `@playplus/perf` package. ```json { "ci": { "collect": { "numberOfRuns": 3 }, "assert": { "assertions": { "core-web-vitals": "error", "largest-contentful-paint": ["error", { "maxNumericValue": 2500 }], "cumulative-layout-shift": ["error", { "maxNumericValue": 0.1 }], "interactive": ["warn", { "maxNumericValue": 3800 }] } } } } Developer Checklist ■ Are all non-critical third-party scripts or tasks wrapped in playperf.defer() ? Is the code for my route being dynamically imported (code-splitting)? Are all images lazy-loaded and served in modern formats (e.g., WebP) with explicit width and height attributes to prevent CLS? Have I used skeleton loaders for data-heavy components instead of spinners? For long lists, am I using a virtualized scrolling solution? Are pure components memoized using React.memo or Angular's OnPush change detection strategy? Have I analyzed the bundle size impact of any new dependencies I've added? Does my feature pass the Lighthouse CI performance budget checks? Performance Checklist ■ Development ■ All non-critical tasks use playperf.defer() Components use OnPush change detection where appropriate All ngFor loops have trackBy functions Images are lazy loaded with explicit

dimensions Routes are lazy loaded Heavy computations are measured with playperf.measure() Testing ■ Performance monitoring is initialized in app component Component render times are within 16ms budget Bundle size is under 250KB limit Lighthouse scores meet performance budgets Long tasks are eliminated Production ■ Core Web Vitals are monitored Performance metrics are logged Bundle analysis is performed CDN is configured for static assets Gzip compression is enabled Recommended Monitoring Tools ■ For a full view of real-world performance, we recommend integrating with: Web Vitals JS Library : To send Core Web Vitals from the browser to your analytics or logging provider. Sentry Performance : To track slow transactions and identify main thread stalls. SpeedCurve or New Relic : For in-depth Real User Monitoring (RUM) analysis over time, across geographies and devices. Minimum Enforcement Thresholds ■ All Play+ applications in the Golden Path are held to these baseline standards, which are enforced automatically in the CI/CD pipeline. Metric Target LCP (Largest Contentful Paint) < 2.5s INP (Interaction to Next Paint) < 200ms CLS (Cumulative Layout Shift) < 0.1 JS Bundle Size (gzipped) < 250KB Main Thread Long Tasks (>50ms) 0